2)

TOWARDS A LIQUID COMPILER

by

Stephen Brooks Davis

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October, 1993

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science, October 5, 1993

Certified by [illegible]
Thomas F. Knight Jr., Principal Research Scientist

Accepted by [illegible]
F.R. Morgenthaler
Chair, Department Committee of Graduate Students

Stephen Brooks Davis

Abstract

Liquid is an architecture for supporting the parallelism of mostly-functional sequential programs. Under Liquid, memory becomes a database, and the program becomes a series of transactions operating on this database. A transaction is free to reference the database, but can not write to it until it commits. By constraining the commits to occur in the same order as if executing sequentially, the memory maintains sequential consistency. By allowing a transaction to roll back, the compiler does not need to be conservative in the presence of aliasing. This work describes progress towards building a compiler to parallelize Scheme programs. In particular, the Liquid abstract machine (LAM), and the transformations from Scheme to LAM are described.

Thesis Supervisor: *Thomas F. Knight Jr.*
Title: *Principal Research Scientist*

Liquid

Liquid is an architecture for supporting the parallelization of sequential programs. In particular, its goal is to provide architecture support for parallelizing mostly functional programs. A mostly functional program is one which makes limited use of side-effects to shared memory objects. A side-effect occurs when a memory location is written to after it has already been initialized with an initial value.

If side-effects do not occur, then execution of different sections of code can proceed as soon as data dependencies are satisfied. The order in which these side-effect free sections execute is not important. However, once side-effects are permitted, sections executing in different order can produce different results. Since order is important, a compiler trying to parallelize a sequential program has to be conservative in order to guarantee that the parallelized program generates the same results as the sequential program.

In particular, the compiler has to guarantee that a value is not read until the correct value is written. This determination is complicated by the presence of aliasing. Aliasing occurs when a memory object can be manipulated by more than one name. For example, a particular memory object is referred to by names **A** and **B**. If a side-effect is performed on **B**, then **A** is side-effected as well. Therefore, the compiler needs to know that modifying **B** will result in a modification of **A**. But, this can not always be determined statically. If the compiler can not prove that **A** and **B** are never aliases for each other, it has to assume that they are.

This assumption guarantees that in the cases where **A** and **B** are aliased, the correct result is generated. However, when **A** and **B** are independent, the assumption prohibits taking advantage of their independence to gain parallelism. What Liquid provides is the ability to change assumptions. Under Liquid, when it can not be proved that **A** and **B** are always aliased, assume they are not. If this assumption is proved false during run-time, roll back computation to that point and resume.

In order to catch when the assumption has been violated, Liquid treats the memory system as a database. Under this abstraction, program execution becomes transactions operating on the database, and the run-time system guarantees that the database remains consistent.

To support the database abstraction, the sequential program is broken up into transactions. A transaction is a potential computation until it commits. Until committing, the transaction is allowed to reference the database, but can not side-effect it. All side-effects within the transaction are delayed until commit.

Each transaction contains a sequence of transactional reads and writes to the database. Whenever a transactional read is performed, the run-time system records which transaction has performed the read and a place to roll back to in the event that the data read becomes stale. Stale data occurs when a value is read before the correct value has been written. Transactional writes are cached by the run-time system until commit. At commit time, all cached writes actually perform their side-effects on the database.

To guarantee that the parallelized program produces the same result as the sequential one, the transactions are committed in the same order they would occur when executed sequentially. This guarantees that the memory system maintains sequential consistency.

This thesis discusses the preliminary work done towards building a compiler for the Liquid architecture. The language chosen to parallelize is Scheme. It was chosen because it is natural to write mostly functional programs in. Scheme does not provide any constructs for explicit parallelism, nor was the language designed for parallel execution.

The current compiler only takes advantage of procedure parallelism. That is the parallelism available by having multiple procedures executing simultaneously.

Scheme is translated into the Liquid Abstract Machine (LAM) representation. LAM provides the support for the transaction mechanism, and for parallel execution. It is designed to take advantage of non uniform memory access (NUMA) architectures. In particular, the goal is to be able to execute the Liquid abstraction on top of the MBTA architecture. LAM has developed out of UC Berkeley's Threaded Abstract Machine (TAM).

LAM Specifications

LAM Abstractions

The Liquid Abstract Machine (LAM) is based upon the Threaded Abstract Machine (TAM) developed at UC Berkeley. It is an abstract machine representation which takes advantage of non uniform memory access (NUMA) architectures. Under LAM, there is a clear distinction between local and remote storage.

The primary difference between LAM and TAM is the introduction of the transaction abstraction in LAM. Both support the abstractions of code blocks, frames, inlets, threads, and quantums.

The changes made for LAM are as follows:

- LISPish parenthesized syntax
- Objects of type generic have a type and value field
- Array and code block data types added
- Send, receive, read, and write support the transaction abstraction
- Commit instruction added
- Inlets do not fork other inlets

Code blocks are the unit of scheduling in LAM. Each code block contains declarations, a frame template, a set of inlets, and a set of threads. Each instantiation of a code block executes relative to a frame. There is exactly one frame associated with each code block instantiation. All the inlets and threads of an instantiation execute relative to this frame.

A frame is a collection of slots. These slots hold synchronization information, message data, and temporary results. The storage required for a frame is statically determined, but dynamically allocated. A frame resides on a single processing node, and computation relative to the frame occurs on the same node. The frame represents a code block instantiation. In the current implementation, a frame is assigned a processor and a location on that processor which can not change during its lifetime.

The inlets are specialized message handlers. Each inlet has a receive instruction specifying how to interpret the message and in which frame slots to store the message contents. To perform computation on the message, the inlet forks a thread. The message destination is specified by frame pointer and inlet number. The frame pointer specifies the code block, and the inlet number the particular inlet within that code block. Unlike TAM inlets, a LAM inlet can only post a thread, not another inlet.

Threads execute instructions relative to the current frame. Threads can fork other threads, but can not fork inlets. A thread executes from the start of the thread to the end. No instructions are skipped. LAM threads are essentially basic blocks, except that a thread may fork to multiple threads.

There are two vectors for each code block instantiation used for scheduling. These are the remote continuation vector (RCV), and the local continuation vector (LCV). The RCV holds the threads posted by inlets. The LCV holds the threads forked by other threads. Each vectors is a queue. Entries can be added to the RCV at any time, whereas the LCV receives entries only when the instantiation is the current frame.

When an instantiation is scheduled for execution, the RCV is copied to the LCV. The instantiation remains scheduled until there are no more threads on the LCV. The period from when the RCV is copied to when the LCV is empty is a *quantum*. The sizes of the RCV and LCV are statically determined. The RCV's size is equal to the number of inlets, and the LCV's size to the number of threads in the code block.

Posting a thread within an inlet places the thread onto the RCV. Thus a posted thread will not execute until the next quantum. When a thread forks another thread, the forked thread is placed in the LCV. All threads forked are executed within the current quantum. Threads only execute when the frame they execute relative to becomes the current frame. Inlets execute regardless of whether or not their frame is the current frame.

The transaction mechanisms have been added to LAM to ensure that the parallelized sequential program generates the same results as the sequential version. Towards this end, all transactional reads record which transaction performed the read and where to send the rollback message in the event that the read has read stale data. Stale data occurs when the read has read before the correct value has been written. Inlets handle the rollback messages.

Any side-effects to shared objects can not occur until the transaction performing the side-effect commits. To guarantee this, all operations which perform a side-effect on a shared object must be delayed until commit. For transactional writes, the run-time system caches the write until the commit. For a message send which will cause the receiver to perform a side-effect, the delayed send operator needs to be used. A delayed send message is cached until commit.

Lam Data types

The data types supported by LAM are frame-pointer, integer, generic, array, code-block, sync-type, generic-ptr, and int-ptr.

Integers are used to represent numeric integers, Booleans, inlet identifiers, thread identifiers, and transaction identifiers. Inlet identifiers specify the inlet number within the code block. Thread identifiers represent the thread number. A transaction identifier is used by transactional reads and writes, and the commit operation to specify which transaction within a code block is performing the operation.

Generics have two fields: type and value. The generic type field is used to implement data types not provided by LAM. The value field can hold any of the LAM data types except arrays and generics.

The array type allows for the creation of static vectors within the frame. The declaration for arrays is (array type size). Where type is any of the LAM data types except for arrays. Size is the number of elements to allocate.

Code block is used to hold a reference to a code block object. The only LAM operator which can use an object of type code block is falloc. This data type has been added to support Scheme's first class procedures.

The sync-type allows for synchronization within LAM. Synch slots are only referenced at the beginning of a thread. Slots which are of sync-type hold an integer value which specifies the number of times the thread needs to be forked and/or posted before the thread will be allowed to execute. Frame slots of type sync-type are initialized when the frame containing them is created.

Generic-ptr points to a dynamically allocated generic object. Int-ptr points to a dynamically allocated integer.

LAM operations

message handling

```
(receive)
(receive (current_frame slot_1)
         (current_frame slot_2)
                  ...
         (current_frame slot_n))
```

The receive operator receives the message and interprets it based on the types of the frame slots specified. **Receive** can only occur in inlets.

```
(send frame inlet (current_frame value_1)
                  (current_frame value_2)
                           ...
                  (current_frame value_n))
```

Send the message composed of the frame slots **value_1, value_2, ..., value_n** to the inlet **frame/inlet**. **Frame** is a frame pointer specifying the destination frame, and inlet is an integer specifying the specific message handler to execute relative to the frame. The receiving inlet (**frame/inlet)** will have a corresponding receive instruction specifying the same number of frame slots with the same types as in the send instruction.

```
(dsend trans-id frame inlet (current_frame value_1)
      (current_frame value_2) ...
      (current_frame value_n))
```

Dsend is a send which is not sent until the commit instruction for transaction **trans-id** is issued. Delayed sends are used when receipt of the message causes a side-effect to be performed on a shared data object stored in the destination frame.

arrays

Arrays are allocated storage within the frame. Their size is statically declared, and can not be dynamically changed. The only operations which can manipulate arrays are **arr-index** and **arr-set**. These operations manipulate individual elements within the array vector. There are no operations which work on an entire array. (**array type size**) declares a frame slot as type array with **size** elements each of type **type**. The **type** can be any of the LAM types except array.

(arr-index destination array index)

Stores the value read from **array[index]** into the frame slot **destination**. **Array** is a frame slot of type array. **Index** is of type integer. **Destination** must be of the same type as the elements of the vector **array**.

(arr-set array index value)

Stores **value** in **array[index]**. **Array** is a frame slot of type array. **Index** is of type integer. **Value** must be of the same type as the elements of **array**.

generics

LAM generics differ from generics in TAM. A LAM generic has two fields: type and value. The type field is of type integer and represents data types built on top of LAM. The value field of a generic can hold object of any of the LAM types except array and generic. A type field of 0 is reserved to specify a generic of type integer. That is, if the type field contains a 0, then the value field contains an object of type integer. The value field of a generic is not statically typed. The "type" of the value field is determined dynamically based on the contents at run-time. The LAM run-time system does not make use of the information contained in the type field.

(set-type destination type-val)

Sets the type field of the generic **destination** to have the integer value **type-val**.

(set-value destination val)

Sets the value field of the generic **destination** to have the value **val**.

(type destination generic)

Stores in **destination** the type identifier stored in the type field of **generic**. **Destination** is of type integer.

(value destination generic)

Stores in the frame slot **destination** the value stored in the value field of **generic**. The LAM types for **destination** and the contents of the value field of **generic** must match at run-time.

(make-integer destination int)

Make-integer sets the type field of **destination** to integer, and the value field to **int**. **Destination** is of type generic. **Int** has type integer.

memory operations

Storage allocated by halloc or malloc is a typed dynamically created array. References of the form **mem-ptr[offset]** signify the **offset**'th entry in the array. The array indices begin at 0.

Transactional reads provide the memory system with the identification of which transaction is reading, and where to send the rollback message when a dependency violation is detected for this read. A transaction dependency occurs when a committing transaction writes to a location read by a non committed transaction.

The run-time system caches transactional writes until the commit instruction is executed. When the commit occurs, the cached writes side-effect memory.

(malloc type dest size)

Malloc allocates a vector of memory from the same processing node that is executing the **malloc** instruction. **Malloc** allocates a vector of **size** elements of type **type** from the heap. The pointer to the vector is stored in the frame slot **dest**. The **type** specified can be any of the LAM data types except array. **Size** is of type integer. **Dest** must be a pointer type (i.e. if **type** is generic, then **dest** is generic-ptr). The allocated vector may be manipulated by **lread, ltread, lwrite, ltwrite,** and **mfree**. If the request can not be met, **dest** will contain the value 0. **Type** is a constant symbol specifying the LAM data type.

(mfree mem-ptr)

Mfree deallocates the storage pointed to by **mem-ptr. Mem-ptr** points to memory allocated by malloc. **Mem-ptr** is a pointer type.

(halloc type dest-frame dest-inlet size)

Halloc allocates storage for a vector with **size** elements of type **type**. **Halloc** attempts to allocate the storage on the processing node executing the **halloc** instruction. However, if the allocation request can not be satisfied by the current processing node, the memory manager will allocate the memory from a remote node. The inlet specified by **dest-frame** and **dest-inlet** receives the pointer to the memory allocated. If the allocation request can not be met, a value of 0 is sent. **Dest-frame** is of type frame-pointer. **Dest-inlet** is of type integer. **Type** is a constant symbol specifying the LAM data type. The allocated vector can be manipulated by **hread, htread, hwrite, htwrite,** and **hfree.**

(hfree mem-ptr)

Hfree deallocates the storage pointed to by **mem-ptr. Mem-ptr** points to memory allocated by halloc. **Mem-ptr** is a pointer type.

(lread type dest mem-ptr offset)

Lread performs a non-transactional reference to memory allocated by malloc. **Lread** reads the value stored in **mem-ptr[offset]** and stores it in the frame slot **dest**. **Type** specifies the LAM data type of the object stored at **mem-ptr[offset]**. **Dest** has type **type**. **Offset** has type integer. **Type** is a constant symbol specifying the LAM data type.

(ltread type trans-id dest mem-ptr offset roll-frame roll-inlet)

Ltread performs a transactional reference to memory allocated by malloc. **Ltread** reads the value stored in **mem-ptr[offset]** and stores it in the frame slot **dest**. **Type** specifies the LAM data type which is stored at **mem-ptr[offset]**. **Dest** must be of type **type**. **Roll-frame** and **roll-inlet** specify the inlet to call if a rollback is required because the **ltread** has read stale data. **Roll-frame** is of type frame-pointer. **Roll-inlet** is of type integer. **Trans-id** is of type integer and specifies the transaction identifier. Together **roll-frame** and **trans-id** uniquely define the transaction performing the read. **Offset** has type integer. **Type** is a constant symbol specifying the LAM data type.

(lwrite type mem-ptr offset value)

Lwrite performs a non-transactional side-effect on memory allocated by malloc. **Lwrite** stores **value** in **mem-ptr[offset]**. **Type** specifies the LAM data type of **mem-ptr[offset]**. **Value** must be of type **type**. **Type** is a constant symbol specifying the LAM data type.

(ltwrite type trans-id mem-ptr offset value)

Ltwrite performs a transactional side-effect on memory allocated by malloc. **Ltwrite** stores **value** in **mem-ptr[offset]**. **Type** specifies the LAM data type of **mem-ptr[offset]**. **Value** must be of type **type**. **Type** is a constant symbol specifying the LAM data type.

(hread type dest-frame dest-inlet mem-ptr offset)

Hread performs a non-transactional split-phase reference from memory allocated by halloc. **Dest-frame** and **dest-inlet** specify the inlet to send the value of type **type** stored at memory location **mem-ptr[offset]**. **Offset** and **dest-inlet** have type integer. **Mem-ptr** is a pointer type.

Dest-frame is of type frame-pointer. **Type** is a constant symbol specifying the LAM data type.

(htread type trans-id dest-frame dest-inlet mem-ptr offset roll-frame roll-inlet)

Htread performs a transactional split-phase reference from memory allocated by halloc. **Dest-frame** and **dest-inlet** specify the inlet to send the value of type **type** stored at memory location **mem-ptr[offset]**. **Roll-frame** and **roll-inlet** specify the rollback point for this read. Together **roll-frame** and **trans-id** uniquely determine the transaction performing the read. **Trans-id, dest-inlet, offset**, and **roll-inlet** are of type integer. **Dest-frame** and **roll-frame** are of type frame-pointer. **Mem-ptr** is a pointer type. **Type** is a constant symbol specifying the LAM data type.

(hwrite type mem-ptr offset value)

Hwrite performs a non-transactional side-effect on memory allocated by halloc. **Hwrite** stores **value** in **mem-ptr[offset]**. **Type** specifies the LAM data type of **mem-ptr[offset]**. **Value** must be of type **type**. **Type** is a constant symbol specifying the LAM data type.

(htwrite type trans-id mem-ptr offset value)

Htwrite performs a transactional side-effect on memory allocated by halloc. **Htwrite** stores **value** in **mem-ptr[offset]**. **Type** specifies the LAM data type of **mem-ptr[offset]**. **Value** must be of type **type**. **Type** is a constant symbol specifying the LAM data type.

frame operations

(falloc return-frame return-inlet code-block)

Allocates a frame for the code block specified by **code-block**. **Code-block** can either be a constant identifier for a code block, or a frame slot of type code-block. A frame will be allocated on a processing node, and inlet 0 of the code-block will be scheduled. Inlet 0 initializes the synchronization variables in the frame, and returns the pointer to the newly created frame, to the inlet specified by **return-frame** and **return-inlet**. **Return-frame** has type frame-pointer. **Return-inlet** is an integer.

(ffree frame-ptr)

Frees the frame storage for **frame-ptr**. **Frame-ptr** is of type frame-pointer.

commit mechanisms

(commit trans-id)

Perform a commit on the transaction identified by the current_frame pointer and **trans-id**.

(reference ref-frame trans-id ident roll-frame roll-inlet)

Adds a read dependency for a reference made by the transaction specified by **ref-frame** and **trans-id**. **Ident** is used as the "address" of the reference. This instruction is for use when a object stored in the frame can be referenced by other code blocks. In the event this reference leads to a rollback, the rollback inlet specified by **roll-frame** and **roll-inlet** will be called. **Ref-frame** and **roll-frame** have type frame-pointer. **Trans-id**, **ident**, and **roll-inlet** have type integer.

(side-effect frame-ptr trans-id ident)

This operations is used in the commit section of a transaction when a shared data structure implemented on top of LAM is side-effected.

branching

(post thread-number)

Schedules the thread **thread-number** to execute in the next quantum. The thread is queued in the RCV.

(fork thread-number)

Schedules a thread for execution within the current quantum. The thread is queued in the LCV.

(switch conditional thread-true thread-false)

If **conditional** is true fork **thread-true**, otherwise fork **thread-false**.

markers

(stop)

Marks the end of a thread or inlet. If there are other threads in the LCV, then the head of the LCV becomes the executing thread, otherwise the quantum ends and another code block is scheduled for execution.

(sync sync-slot)

Marks the beginning of a thread. **Sync-slot** specifies the number of times this thread must be scheduled for execution before it is allowed to compute. In the case where it must only be scheduled to compute once, this instruction may be omitted from the thread.

data movement

(move destination source)

Copy the contents of frame slot **source** to frame slot **destination. Source** and **destination** must be of the same LAM type, and cannot be of type array.

arithmetic operations

(*prim-op* destination arg1 arg2)

Prim-op is one of the following operations:

int == integer
ptr == generic-ptr or int-ptr

LAM op	operation
eq	int or ptr equality
ge	int or ptr greater than or equal
gt	int or ptr greater than
lt	int or ptr less than
le	int or ptr less than or equal
add	int or ptr addition
sub	int or ptr subtraction
mul	integer multiply
div	integer divide
and	integer logical and
or	integer logical or
max	integer maximum
min	integer minimum

The values for true and false are inherited from the C implementation of LAM.

LAM false: integer 0
LAM true : Anything other than integer 0

Scheme to LAM

Subset of Scheme implemented on top of LAM

Data Types
Integer
Null
Cons
Vector
Procedure (the rest operator has not been implemented)

Operations
+, -, *, /, and, or, =, >=, <=, >, <, max, min
cons, car, cdr, pair?, set-car!, set-cdr!, null?
vector?, vector-length, vector-ref, make-vector, vector-set!
set!, and variable references.

How Scheme is partitioned into LAM

Every Scheme procedure becomes a LAM code block. Parallelism occurs whenever a procedure call is made. Since each code block is independently scheduled, each code block can execute in parallel.

Under the current compiler implementation, procedure parallelism is the only form of parallelism utilized.

Implementation of Scheme Data types

Integers

Since integer operations are built into LAM no special representation is needed to support basic Scheme integers. There is currently no support for bignums.

LAM implements Boolean with the integer data type. This differs from TAM which has an explicit Boolean data type.

primitive operations

Scheme expression: (prim-op **op1 op2**)

Where prim-op is one of the following Scheme operators:
+, -, *, /, =, >=, <=, <, >, max, min, and, or

The corresponding LAM operators are as follows:

Scheme op	**LAM op**
+	add
-	sub
*	mul
/	div
=	eq
>=	qe
<=	le
<	lt
>	gt
max	max
min	min
and	and
or	or

Frame declarations

```
(temp_int1  integer)
(temp_int2  integer)
(temp_val   integer)
(temp_op1   generic)
(temp_op2   generic)
(temp_res   generic)
```

Generated code

```
(define-thread eval-primop
   (fork eval-op1)
   (fork eval-op2)
   (stop))
```

thread *eval-op1*
  Evaluates **op1** to a value of type generic. The computed value is stored in the frame slot temp_*op1*, and then forks thread *compute-op*.

thread *eval-op2*
  Evaluates **op2** to a value of type generic. The computed value is stored in the frame slot temp_*op2*, and then forks thread *compute-op*.

```
(define-thread compute-op
   (sync (current_frame sync_i))
   (value (current_frame temp_int1)
          (current_frame temp_op1))
   (value (current_frame temp_int2)
          (current_frame temp_op2))
   (lam-prim-op (current_frame temp_val)
                (current_frame temp_int1)
                (current_frame temp_int2))
   (make-integer (current_frame temp_res)
                 (current_frame temp_val))
   (fork cont))
```

Sync_*i*'s initial value is 2. Lam-prim-op is the LAM mapping of the Scheme prim-op.

Thread *eval-primop* causes both operands to begin computing. The operands are evaluated to generic integers and will be in frame slots temp_*op1* and temp_*op2* when thread *compute-op* begins execution.

Thread *compute-op* performs the primitive arithmetic operation. It converts the two generic integers into LAM integers, performs the specified arithmetic operation, and converts the resulting integer to a generic integer. The generic integer is returned in frame slot temp_*res* to the thread *cont*.

Null

An object of type null is a generic whose type field is the integer value null-tag and value field is undefined. Objects of type null are created by the Scheme expression: '().

null?

Scheme expression: (null? **expr**)

Frame declaration

```
(temp_type integer)
(temp_null generic)
(temp_res  generic)
```

Generated code

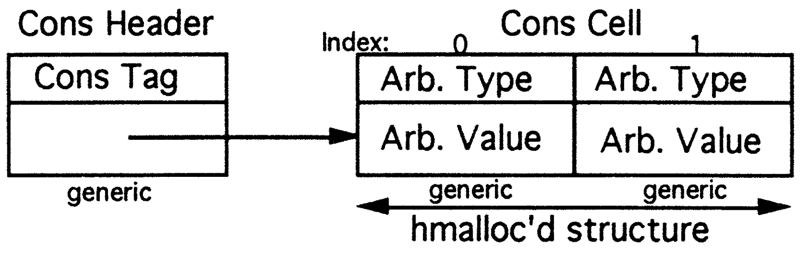
thread *eval-expr*
Computes **expr** to a generic, stores it in temp_*null*, and forks thread *null?*.

```
(define-thread null?
   (type (current_frame temp_type)
         (current_frame temp_null))
   (eq (current_frame temp_type)
         (current_frame temp_type) null-tag)
   (make-integer (current_frame temp_res)
         (current_frame temp_type))
   (fork cont)
   (stop))
```

Thread *null?* gets the type of the generic temp_*null*, comparing it to the null type tag. The Boolean result is converted into an integer and returned to thread *cont* in the frame slot temp_*res*.

Cons

The cons cell is made up of two parts: a generic of type cons (Cons Header), and a heap allocated LAM vector containing two generics.



The Scheme operators for manipulating cons cells are cons, car, cdr, pair?, set-car!, and set-cdr!.

The LAM code generated for each of these operators is as follows:

cons

Scheme expression: (cons **car-exp cdr-exp**)

Frame declarations

```
(temp_ptr    generic-ptr)
(temp_header generic)
(sync_i      sync-type)
(temp_car    generic)
(temp_cdr    generic)
```

Generated code

```
(define-thread make-cons-cell
   (halloc generic current_frame get_ptr_inlet 2)
   (fork compute_car_thread)
   (fork compute_cdr_thread)
   (stop))
```

thread *compute_car_thread*
Evaluates **car-exp** to a value of type generic. The computed value is stored in the frame slot temp_*car*, and then thread *make_cell_thread* is forked.

thread *compute_cdr_thread*
Evaluates **cdr-exp** to a value of type generic. The computed value is stored in the frame slot temp_*cdr*, and then thread *make_cell_thread* is forked.

```
(define-inlet get_ptr_inlet
   (receive (current_frame temp_ptr))
   (post make_cell_thread)
   (stop))

(define-thread make_cell_thread
   (sync (current_frame sync_i))
   (set-type (current_frame temp_header) cons-tag)
   (set-value (current_frame temp_header)
              (current_frame temp_ptr))
   (hwrite generic  (current_frame temp_ptr) 0
                 (current_frame temp_car))
   (hwrite generic  (current_frame temp_ptr) 1
                 (current_frame temp_cdr))
   (fork cont)
   (stop))
```

Sync_*i*'s initial value is 3. It is posted by *get_ptr_inlet* and forked twice: once each for the car expression and cdr expression.

Make-cons-cell dynamically allocates a vector of two generics which serves as a cons cell. After allocated the cons cell, it forks off computation to evaluate the car and cdr expressions.

Inlet *get_ptr_inlet* receives the cons cell allocated in thread *make-cons-cell*.

The thread *make_cell_thread* begins executing once the car and cdr expressions have evaluated to a value, and storage has been allocated for the cons cell. The threads purpose is to initialize the cons cell. Initialization involves setting the generic holding the cons cell header, temp_*header*, to type cons with value containing the pointer to the cons cell.

The cons cell temp_*header* is returned to thread *cont*.

car

Scheme expression: (car **cons-expr**)

Frame declarations

```
(temp_ptr     generic-ptr)
(temp_cons    generic)
(temp_res     generic)
```

Generated code

thread *eval-car*

Evaluates **cons-expr** to a generic value, storing the result in temp_*cons*, and then forks thread *get_car*.

```
(define-thread get_car
   (value (current_frame temp_ptr)
          (current_frame temp_cons))
   (htread generic trans_x current_frame car_inlet
           (current_frame temp_ptr) 0 current_frame
           car_rollback_inlet)
   (stop))

(define-inlet car_inlet
   (receive (current_frame temp_res))
   (post cont)
   (stop))

(define-inlet car_rollback_inlet
   (receive)
   (post get_car)
   (stop))
```

Thread *get_car* gets the pointer to the cons cell from the cons header temp_*cons*. It then performs a transactional read to get the car portion of the cell.

Inlet *car_inlet* receives the generic value representing the car, and returns it in temp_*res* to the thread *cont*.

In the event that the value read from the cons cell is stale, the inlet *car_rollback_inlet* causes the car portion to be read again. *Car_rollback_inlet* is the rollback point

cdr

Scheme expression: (cdr **cons-expr**)

<u>Frame declarations</u>

```
(temp_ptr  generic-ptr)
(temp_cons generic)
(temp_res  generic)
```

<u>Generated code</u>

thread *eval-cdr*
Evaluates **cons-expr** to a generic value, storing the result in temp_*cons*, and then forks thread *get_cdr*.

```
(define-thread get_cdr
   (value (current_frame temp_ptr)
          (current_frame temp_cons))
   (htread generic trans_x current_frame cdr_inlet
           (current_frame temp_ptr) 1 current_frame
           cdr_rollback_inlet)
   (stop))

(define-inlet cdr_inlet
   (receive (current_frame temp_res))
   (post cont)
   (stop))

(define-inlet cdr_rollback_inlet
   (receive)
   (post get_cdr)
   (stop))
```

Thread *get_cdr* gets the pointer to the cons cell from the cons header temp_*cons*. It then performs a transactional read to get the cdr portion of the cell.

Inlet *cdr_inlet* receives the generic value representing the cdr, and returns it in temp_*res* to the thread *cont*.

In the event that the value read from the cons cell is stale, the inlet *cdr_rollback_inlet* causes the car portion to be read again. *Cdr_rollback_inlet* is the rollback point

pair?

Scheme expression: (pair? **cons-expr**)

Frame declarations

```
(temp_type   integer)
(temp_header generic)
(temp_res    generic)
```

Generated code

thread *eval-cons-expr*
Computes **cons-expr** to a generic value, storing it in temp_*header*, and forks thread *pair?*.

```
(define-thread pair?
   (type (current_frame temp_type)
         (current_frame temp_header))
   (eq (current_frame temp_type)
       (current_frame temp_type) cons-tag)
   (make-integer (current_frame temp_res)
                 (current_frame temp_type))
```

```
    (fork cont)
    (stop))
```

Thread *pair?* gets the type of the generic temp_*header*, comparing it to the cons type tag. The Boolean result is converted into an integer and returned to thread *cont* in the frame slot temp_*res*.

set-car!

Scheme expression: (set-car! **cons-cell expression**)

Frame declarations

```
(sync_i      sync-type)
(temp_ptr    generic-ptr)
(temp_header generic)
(temp_val    generic)
(temp_res    generic)
```

Generated code

```
(define-thread car!
   (fork eval-cons-cell)
   (fork eval-expression)
   (stop))
```

thread *eval-cons-cell*
Computes the value of the **cons-cell** argument, leaves the value in the frame slot temp_*header*, and forks *side-effect-car!*.

thread *eval-expression*
Computes the value of the **expression** argument, leaves the value in the frame slot temp_*val*, and forks *side-effect-car!*.

```
(define-thread side-effect-car!
   (sync (current_frame sync_i))
   (value (current_frame temp_ptr)
          (current_frame temp_header))
   (htwrite generic trans_x (current_frame temp_ptr)
            0 (current_frame temp_val))
   (move (current_frame temp_res)
         (current_frame temp_val))
   (fork cont)
   (stop))
```

Sync_i's initial value is 2.

The thread *car!* causes the **cons-cell** and **expression** arguments to be evaluated. The cons cell is stored in temp_*header* and the new car value in temp_*val*.

Thread *side-effect-car!* gets the pointer to the cons cell from the cons header, transactionally writes the new car value, and returns to the thread *cont* the new car value.

set-cdr!

Scheme expression: (set-cdr! **cons-cell expression**)

Frame declarations

```
(sync_i        sync-type)
(temp_ptr      generic-ptr)
(temp_header   generic)
(temp_val      generic)
(temp_res      generic)
```

Generated code

```
(define-thread cdr!
   (fork eval-cons-cell)
   (fork eval-expression)
   (stop))
```

thread *eval-cons-cell*
Computes the value of the **cons-cell** argument, leaves the value in the frame slot temp_*header*, and forks *side-effect-cdr!*.

thread *eval-expression*
Computes the value of the **expression** argument, leaves the value in the frame slot temp_*val*, and forks *side-effect-cdr!*.

```
(define-thread side-effect-cdr!
   (sync (current_frame sync_i))
   (value (current_frame temp_ptr)
          (current_frame temp_header))
   (htwrite generic trans_x (current_frame temp_ptr)
            1 (current_frame temp_val))
   (move (current_frame temp_res)
         (current_frame temp_val))
   (fork cont)
   (stop))
```

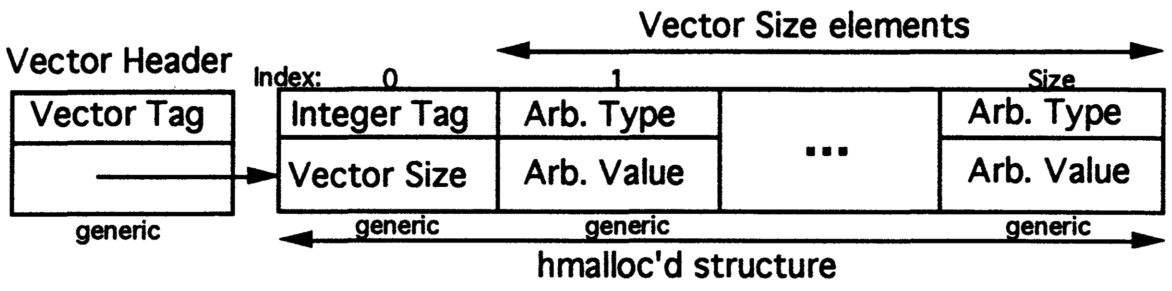
Sync_*i*'s initial value is 2.

The thread *car!* causes the **cons-cell** and **expression** arguments to be evaluated. The cons cell is stored in temp_*header* and the new car value in temp_*val*.

Thread *side-effect-cdr!* gets the pointer to the cons cell from the cons header, transactionally writes the new car value, and returns to the thread *cont* the new car value.

Vector

A vector consists of a generic of type vector (Vector Header), and a heap allocated LAM vector.



The Scheme operators for manipulating vectors are: vector?, make-vector, vector-length, vector-ref, and vector-set!.

vector?

Scheme expression: (vector? **expression**)

Frame declarations

```
(temp_type integer)
(temp_res  generic)
```

Generated code

thread *eval-vector-exp*
Computes **expression** to a generic value, storing it in temp_*header*, and forks thread *vector?*.

```
(define-thread vector?
   (type (current_frame temp_type)
         (current_frame temp__header))
   (eq (current_frame temp_type)
       (current_frame temp_type) vector-tag)
   (make-integer (current_frame temp_res)
                 (current_frame temp_type))
   (fork cont)
   (stop))
```

Thread *vector?* gets the type of the generic temp_*header*, comparing it to the vector type tag. The Boolean result is converted into an integer and returned to thread *cont* in the frame slot temp_*res*.

make-vector

Scheme expression: (make-vector **int-exp**)

Frame declarations

```
(temp_int        integer)
(temp_size       generic)
(temp_alloc_size integer)
(temp_ptr        generic-ptr)
(temp_header     generic)
```

Generated code

thread *eval-int-exp*
Computes **int-exp** to an integer generic, stores it in temp_*size*, and forks thread *make-vector*.

```
(define-thread make-vector
   (value (current_frame temp_int)
          (current_frame temp_size))
   (add (current_frame temp_alloc_size)
        (current_frame temp_int) 1)
   (halloc generic current_frame get_ptr_inlet
          (current_frame temp_alloc_size))
   (stop))

(define-inlet get_ptr_inlet
   (receive (current_frame temp_ptr))
   (post set_up_vector)
   (stop))

(define-thread set_up_vector
   (set-type (current_frame temp_header) vector-tag)
   (set-value (current_frame temp_header)
              (current_frame temp_ptr))
   (hwrite generic (current_frame temp_ptr) 0
          (current_frame temp_size))
   (fork cont)
   (stop))
```

Thread *make-vector* allocates a vector of storage one larger than requested in **int-exp**. The extra vector slot stores the size of the vector. The allocated vector storage is returned to inlet *get_ptr_inlet*, and initialized in thread *set_up_vector*.

In thread *set_up_vector* the vector header is set to have vector type and the pointer to the vector storage as its value. In the first slot of the vector storage, the vector size is stored. *set_up_vector returns* the vector header temp_*header* to thread *cont*.

The write in the thread *set_up_vector* is not transactional since the vector object is not shared at creation.

vector-length

Scheme expression: (vector-length **vector-exp**)

<u>Frame declarations</u>

```
(temp_ptr    generic-ptr)
(temp_header generic)
(temp_size   generic)
```

<u>Generated code</u>

thread *eval-vector-exp*
  Computes **vector-exp** to a value which is stored in temp_*header*, then thread *vector-length* is forked.

```
(define-thread vector-length
   (value (current_frame temp_ptr)
          (current_frame temp_header))
   (hread generic current_frame have_size_inlet
          (current_frame temp_ptr) 0)
   (stop))

(define-inlet have_size_inlet
   (receive (current_frame temp_size))
   (post cont)
   (stop))
```

The thread *vector-length* gets the pointer to the dynamically allocated vector, and does a non-transactional read to get the vector length information. The read is non-transactional because the size of a vector can not change after its creation.

Inlet *have_size_inlet* receives the vector size read in thread *vector_length* and returns it to thread *cont*.

vector-ref

Scheme expression: (vector-ref **vector index**)

<u>Frame declarations</u>

```
(sync_i      sync-type)
(temp_ptr    generic-ptr)
(temp_vector generic)
(temp_int    integer)
(temp_res    generic)
```

Generated code

```
(define-thread vector-ref
   (fork evaluate-vector)
   (fork evaluate-index)
   (stop))
```

thread *evaluate-vector*
Computes the **vector** argument, stores the value in temp_*vector*, and forks the thread *do-vector-ref*.

thread *evaluate-index*
Computes the **index** argument, stores the value in temp_*index*, and forks the thread *do-vector-ref*.

```
(define-thread do-vector-ref
   (sync (current_frame sync_i))
   (value (current_frame temp_ptr)
          (current_frame temp_vector))
   (value (current_frame temp_int)
          (current_frame temp_index))
   (add (current_frame temp_int)
        (current_frame temp_int) 1)
   (htread generic trans_x current_frame vec-ref-inlet
           (current_frame temp_ptr)
           (current_frame temp_int)
           current_frame vec-rollback-inlet)
   (stop))
```

Sync_*i*'s initial value is 2.

```
(define-inlet vec-ref-inlet
   (receive (current_frame temp_res))
   (fork cont)
   (stop))

(define-inlet vec-rollback-inlet
   (receive)
   (post do-vector-ref)
   (stop))
```

Thread *do-vector-ref* takes the pointer to the vector structure from the vector header temp_*vector*, computes the index to reference, and performs a transactional read to get the vector entry.

Inlet *vec-ref-inlet* receives the entry contained at **vector[index]** and returns it in frame slot temp_*res* to thread *cont*.

In the event a roll back occurs because of the vector reference, the inlet *vec-rollback-inlet* receives the rollback request and posts thread *do-vector-ref* to read in the new vector value.

vector-set!

Scheme expression: (vector-set! **vector index value**)

Frame declarations

```
(sync_i      sync-type)
(temp_idx    integer)
(temp_index  integer)
(temp_ptr    generic-ptr)
(temp_val    generic)
(temp_res    generic)
```

Generated code

```
(define-thread vector-set!
    (fork evaluate-vector)
    (fork evaluate-index)
    (fork evaluate-value)
    (stop))
```

thread *evaluate-vector*
Computes **vector,** stores the vector object in temp_*vector*, and forks the thread *do-vector-set!*.

thread *evaluate-index*
Computes the **index** argument, stores the value in temp_*index*, and forks the thread *do-vector-set!*.

thread *evaluate-value*
Computes **value**, storing it in temp_*val*, and forks the thread *do-vector-set!*.

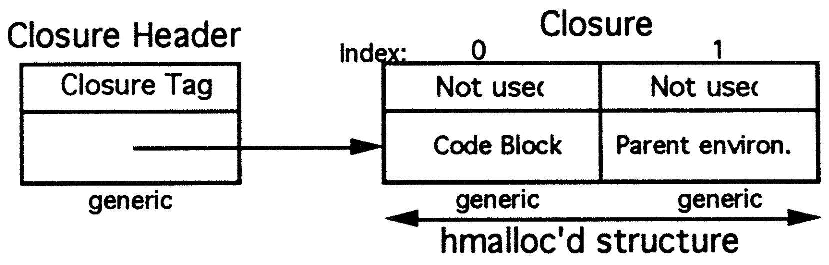
```
(define-thread do-vector-set!
    (sync (current_frame sync_i))
    (add (current_frame temp_idx)
         (current_frame temp_index) 1)
    (value (current_frame temp_ptr)
           (current_frame temp_vector))
    (htwrite generic trans_x
             (current_frame temp_ptr)
             (current_frame temp_idx)
             (current_frame temp_val))
    (move (current_frame temp_res)
          (current_frame temp_val))
    (fork cont)
    (stop))
```

Sync_*i*'s initial value is 3.

Thread *do-vector-set!* places the new value of **value** in **vector[index]**. The thread can execute once temp_*index*, temp_*vector*, and temp_*val* have been initialized by the threads *evaluate-index*, *evaluate-vector*, and *evaluate-value* respectively. The new value is returned in temp_*res* to thread *cont*.

Closures

Closures represent Scheme procedure objects. Closures are first class objects in Scheme. Once a closure is created it is never side-effected.



Frame declarations

```
(temp_ptr     generic-ptr)
(temp_res     generic)
(temp_block   generic)
(temp_enc_env generic)
```

Generated code

```
(define-thread alloc-closure
   (halloc generic current_frame have_closure 2)
   (stop))

(define-inlet have-closure
   (receive (current_frame temp_ptr))
   (post initialize_closure))
   (stop))

(define-thread initialize-closure
   (set-type (current_frame temp_res) closure-tag)
   (set-value (current_frame temp_res)
              (current_frame temp_ptr))
   (set-value (current_frame temp_block) code_block)
   (set-value (current_frame temp_enc_env)
              current_frame)
   (hwrite generic (current_frame temp_ptr) 0
          (current_frame temp_block))
```

```
    (hwrite generic (current_frame temp_ptr) 1
            (current_frame temp_enc_env))
    (fork cont)
    (stop))
```

Thread *initialize-closure* creates a closure object. The closure header generic's type is set to closure-tag, and it's value is a pointer to the dynamically allocated closure array. The array consists of 2 generics, the first contains the code block which computes the procedure, and the second contains the environment where the procedure was created. The closure is returned in temp_*res* to thread *cont*.

Calling conventions

For a procedure taking N arguments, inlet 6 receives the first argument, inlet 7 the second, and so on up to inlet N+5. Inlet N+6 receives the continuation. The continuation specifies the frame and inlet number to send the resultant value to.

The first argument is stored in the environment frame at index offset 0, second argument at index offset 1, and so on up to index offset N. Each argument is passed as a generic.

Argument inlet

For 1 <= *j* <= N

```
(define-inlet j+5
     (receive (current_frame arg_j))
     (arr-set (current_frame array_0)
             j-1
             (current_frame arg_j))
     (post kj)
     (stop))
```

Where k_j is the thread to execute in response to receiving this argument. The thread k_j depends on the data dependencies. However, since the current compiler does not generate a data dependency graph, all arguments fork to thread 1.

Continuation inlet

```
(define inlet N+6
     (receive (current_frame cont_fp_0)
              (current_frame cont_inlet_0))
     (post kc)
     (stop))
```

Where k_C is the thread containing the send to the continuation.

Since each argument is a generic, there is added overhead for referencing the value field from the generic for use in computation. However, this overhead is necessary to support a general call mechanism for Scheme.

By allocating independent inlets for each argument, it becomes possible to begin computation within the procedure before all arguments have arrived. For correctness, the program must synch on the argument variable only before it's first reference. The independent inlets also provides for the parallel evaluation of the arguments. Since Scheme does not place an ordering on the order in which arguments evaluate, they can all evaluate in parallel. However, from the point of view of memory accesses, compilation specifies an arbitrary ordering for commit passing.

Environment specifications

Since Scheme is a statically scoped language, the size of each environment frame is known statically. Since the subset of Scheme implemented does not support dynamic creation of environment entries, the static frame information remains constant during execution. Also, since the subset does not provide first class environments, every environment reference can be statically determined.

Because each frame's size is fixed and statically determined, storage is statically assigned within the frame allocated to compute the procedure. Environments are implemented as a vector of generics.

In each code block, inlets 2 through 5 are set aside for the environment. An environment parent is specified by the frame pointer containing the parent environment frame.

Inlet 2 sends to a child environment frame all of its parents. These parents consist of the current environment's parents, plus a pointer to the current environment.

Inlet 3 receives all of the parents for the current environment frame.

Inlet 4 performs a variable lookup within the current environment frame.

Inlet 5 performs a side effect on a variable within the current environment frame.

For the purpose of counting frames, the global environment is at level 0, it's child at level 1, and so on.

Specifications

For an environment frame at level **N** with **M** entries in the frame.

Frame declarations
```
(frame_ptr_1     frame-pointer)
(inlet_ref_1     integer)
(frame_ptr_2     frame-pointer)
(inlet_ref_2     integer)
(index_0         integer)
(index_1         integer)
(temp_0          generic)
(arg_0           generic)
(roll_frame_0    frame-pointer)
(roll_inlet_0    integer)
(trans_0         integer)
(trans_1         integer)
(parent_1        frame-pointer)
(parent_2        frame-pointer)
                 :
(parent_N        frame-pointer)
(array_0         (array generic M))
```

1) **Sending parent list to child environment frame**

inlet
```
(define-inlet 2
    (receive (current_frame frame_ptr_1)
             (current_frame inlet_ref_1))
    (send (current_frame frame_ptr_1)
          (current_frame inlet_ref_1)
          (current_frame parent_1)
          (current_frame parent_2)
                      :
          (current_frame parent_N)
          current_frame)
    (stop))
```

This inlet is called when a child of this environment frame is created. The child will be at level **N**+1 within the environment. The receive takes the frame (**frame_ptr_1**) and inlet number (**inlet_ref_1**) specifying the inlet to send the **N**+1 parent frame pointers.

2) **Receiving parent list from parent environment frame**

inlet

```
(define-inlet 3
    (receive (current_frame parent_1)
             (current_frame parent_2)
                        :
             (current_frame parent_N))
    (stop))
```

Since this environment is at level **N** within the environment hierarchy, it receives **N** parent frame pointers.

3) **Variable reference from environment frame**

inlet

```
(define-inlet 4
    (receive (current_frame frame_ptr_2)
             (current_frame inlet_ref_2)
             (current_frame index_0)
             (current_frame roll_frame_0)
             (current_frame roll_inlet_0)
             (current_frame trans_0))
    (arr-index (current_frame temp_0)
               (current_frame array_0)
               (current_frame index_0))
    (reference (current_frame frame_ptr_2)
               (current_frame trans_0
               (current_frame index_0)
               (current_frame roll_frame_0)
               (current_frame roll_inlet_0))
    (send (current_frame frame_ptr_2)
          (current_frame inlet_ref_2)
          (current_frame temp_0))
    (stop))
```

Frame pointer **frame_ptr_2** and inlet number **inlet_ref_2** specify the inlet to return the environment reference. **Index_0** provides the offset within the environment frame. Frame pointer **roll_frame_0** and inlet number **roll_inlet_0** denote the rollback point for this reference. Together frame pointer **frame_ptr_2** and transaction identifier **trans_0** uniquely specify the transaction performing the reference.

The array-index instruction fetches the value of the environment variable stored in the environment frame at index position **index_0**.

The reference instruction logs the read dependency with the run-time system. The index position within the environment frame is used as the frame "address" referenced.

The send returns the environment entry to the reader.

4) **Side effect entry in environment frame**

inlet
```
(define-inlet 5
    (receive (current_frame index_1)
             (current_frame arg_0)
             (current_frame frame_ptr_3)
             (current_frame trans_1))
    (arr-set (current_frame array_0)
             (current_frame index_1)
             (current_frame arg_0))
    (side-effect (current_frame frame_ptr_3)
                 (current_frame trans_1)
                 (current_frame index_1))
    (stop))
```

Index offset **index_1** specifies the environment entry to replace with value **arg_0**. Frame pointer **frame_ptr_3** and transaction identifier **frame_ptr_3** uniquely determine the transaction performing the side-effect.

The arr-set instruction updates the environment entry with the new value.

Side-effect informs the run-time system that a write has occurred on the frame "address" **index_1**.

variable references

Referencing an entry in the local frame:

Frame declarations

```
(temp_res   generic)
```

Array_0 is defined as (array generic M), where M is the number of entries in the environment frame.

Generated code

```
(define-thread reference
   (arr-index (current_frame temp_res)
              (current_frame array_0)
              index_const)
```

```
      (reference current_frame trans_id index_const
                 current_frame rollback_inlet)
      (fork cont)
      (stop))

   (define-inlet rollback-inlet
      (receive)
      (post reference)
      (stop))
```

Thread *reference* accesses the array holding the environment frame information directly with the arr-index instruction. The variable entry is located at *index_const* (an integer constant) within the array array_0. The reference instruction notifies the run-time system about the read dependency on the environment entry.

Inlet *rollback-inlet* is the rollback point for this read.

Referencing an entry in another frame:

Frame declarations

```
   (parent_i  frame-pointer)
   (temp_res  generic)
```

Generated code

```
(define-thread reference
   (send  (current_frame parent_i) 4
          current_frame ref_inlet
          index_const
          current_frame rollback-inlet
          trans_id)
   (stop))

(define-inlet ref-inlet
   (receive (current_frame temp_res))
   (post cont)
   (stop))

(define-inlet rollback-inlet
   (receive)
   (post reference)
   (stop))
```

Thread *reference* sends a message to the frame containing the environment frame requesting the value stored in the frame at index *index_const* (an integer constant). The value will be returned to inlet *ref-inlet*. If this reference leads to rollback, inlet *rollback-inlet* will restart computation at thread *reference*. *Trans_id* is an integer constant specifying the transaction identifier.

Inlet *ref-inlet* returns the value of the variable referenced in the frame slot temp_*res* to thread *cont*.

set!

Scheme expression: (set! **variable value**)

thread *side-effect*
Evaluates **value** to a generic value, stores it in temp_*val*, and forks thread *do-side-effect*.

Frame declarations

```
(parent_j  frame-pointer)
(temp_val  generic)
```

If the environment frame being side-effected does not belong to the procedure doing the side-effect:

```
(define-thread do-side-effect
   (dsend trans_id
          (current_frame parent_j) 5
          index_const (current_frame temp_val)
          current_frame trans_id)
   (stop))
```

otherwise:

```
(define-thread do-side-effect
   (dsend    trans_id
        current_frame 5
        index_const (current_frame temp_val)
        current_frame trans_id)
   (stop))
```

Dsend is used instead of send because environment entries are shared objects, and side-effects to shared object can not be modified until commit. Using a send would violate this protocol.

The thread *do-side-effect* causes the value of an environment entry to be updated once the transaction containing it commits.

Index_const is an integer constant specifying where in the environment frame the variable being side-effected is located.

The last two values of dsend, current_frame and *trans_id*, specify the transaction performing the side-effect. *Trans_id* is an integer constant.

Alternate Environment Implementation

Three environment implementations were considered, before going with *alternate two* described below.

Alternate one

Two code blocks are created for every scheme procedure. One to manage the procedure's environment frame, the other the procedure's computation.

The advantage of this approach is that the storage for computation and for environments can be managed independently. In the event the procedure returns a first class procedure, the environment frame will need to remain in existence after the procedure's computation completes. Thus, the storage required for computation can be freed, leaving the environment storage around.

By separating the environment frame and procedure's computation into separate code blocks, each environment operation requires a message send. Ideally, if the procedure and environment code blocks reside on the same processing node, then the message will be low latency. However, under LAM there is no mechanism for specifying that two code blocks should be instantiated on the same processor. Therefore, environment operations are liable to be expensive in terms of latency.

Alternate two

Combine the management of the environment frame and procedure's computation into one frame. This way the environment frame is guaranteed to reside on the same processor as the one computing the procedure. However, once computation completes, the storage required for computation has to remain for as long as the environment frame is referenced. Thus when computation completes, the frame can not be freed until there are no references to the environment frame.

Alternate three

Combine the management of the environment frame and procedure's computation into one frame, but allow the frame to be partitioned into two parts. As in *alternate two*, the environment frame resides on the same processor as the procedure's computation. But now, the computation portion of the frame can be deallocated when the procedure completes computation.

At first glance *alternate three* appears to be the best choice. As soon as computation is completed, the unnecessary storage is freed to be used by another code block. However, in the current implementation of LAM, once a frame is allocated it can not be moved. Thus even though we can free up unneeded storage, memory fragmentation will prevent this memory from being useful. The freed memory is not in use, but is likely too small to be of use for any other code block. Thus, for the time being *alternate two* has been chosen, by virtue of its being simpler than *alternate three* to implement.

However, in the future it is likely that frames will still be fixed to the same processor they were allocated, but will be able to move within that processors storage. This will clear up the fragmentation problem, making *alternate three* the better choice.

Under all three alternatives, garbage collection will be required to free up the storage allocated for the environment frames. At this time no garbage collector has been implemented.

Liquid transactions

A transaction is a collection of inlets and threads in the same code block whose side-effects are grouped together and cached until the transaction reaches it's commit point. Once a transaction is at the commit point, and holds the commit token, committing takes place. Committing involves sending each of the cached transactional writes to the memory system, as well as sending any delayed sends. After all of the delayed sends and transactional writes have been acknowledged, the commit is sent to the next transaction. The commit is sent to transactions in the same order they would occur if they were executing sequentially on a single processor. Commits pass through the call tree in a depth first manner.

Under the current implementation, each Scheme procedure is mapped into a LAM code block. Because of this a code block can contain multiple transactions, each of which needs to be identified for the memory system. A transaction is uniquely identified by the frame pointer it executes relative to, and a transaction identifier. All transactions within a code block are assigned different transaction identifiers. Transaction identifiers need to be unique within a code block but not across code blocks. That is code block A can have transactions labeled 1,2,…,N, and code block B has 1, 2, …,

M. Because a transaction is identified by it's frame pointer and transaction identifier, there is no ambiguity.

A new transaction is created for each procedure call, at the merging of conditionals where at least one branch of the conditional has created a transaction boundary, and at the end of the code block.

Each thread and inlet within a code block is assigned to exactly one transaction at compile time. Whenever a transactional read or write is performed the transaction performing the operation is identified.

Commit protocol

Every code block has inlet 1 reserved for receiving the commit token. When inlet 1 receives the commit token, and computation is at the commit point, the commit instruction executes causing the commit phase to begin.

The commit phase causes all side effects to become visible to the memory system. At commit all delayed sends and all transactional writes to shared memory occur. The run-time system is responsible for caching these sends and transactional writes. For each delayed send and write, any readers of the locations being side-effected will have to rollback. The only exception is that reads belonging to the same transaction as the writer do not cause rollback.

As a part of the commit, all references made by the committing transaction are removed from all dependency lists.

The following describes what the run-time system does when it executes the commit instruction.

```
<At node where commit is occurring>
For every transactional write and delayed send belonging to
the committing transaction.
  Send the message to make the write or send occur.
  Wait for the acknowledgment from each write or send.
Send the commit on to the successor transaction.

<At node where side effected shared object resided>
For each write (at the node the object resides)
   Update the value of the shared object.
   Send rollback messages to all readers that are not from
     the writer's transaction.
   Wait for ack from all rollback messages sent.
   Send ack back to writer.
```

After all writes and delayed sends have been completed, the commit token is passed onto the next transaction.

Rollbacks

A rollback point is created for each thread which references a shared object. These rollback points are represented by inlets. When a read dependency violation is detected, the memory system sends a rollback message to the rollback inlet. This will cause computation to resume at the thread which read it's data value too early.

A dependency violation occurs when a committing transaction performs a side effect on a shared memory object which later transactions have already referenced.

Missing from LAM

LAM currently does not provide support for control speculation. To offer this support, a method of assigning priorities and to terminate computation needs to be introduced. The priorities are to allow promising computations to occur before less promising ones. The priorities would also enable transactions whose results are used the earliest, to compute their values before computations more distant in the call tree. The ability to kill off computation permits the termination of computations which have been determined to be unnecessary.

The ability to terminate computations will also be necessary for the garbage collection phase. The mechanism currently available is ffree. However, this mechanism is only used to free up frame storage. It does not terminate any computation relative to that frame.

In mul-T and STING they have expressed the benefits of being able to steal computation. In particular, scheduling some computations on the local node. This is particularly useful when the value is needed immediately or for load balancing. However, LAM can not support stealing since it is not possible to move a frame once allocated.

LAM has no notion of scheduling for locality. The falloc mechanism magically picks some processor to allocate the frame on. Once allocated, there are no mechanisms in place to move the frame to another processing node. Although this could be added into the run time system, there would be no mechanism for specifying at the LAM level that two code blocks should execute near each other.

Future Work

Alternate Scheme transformation

The current technique is to compile each Scheme procedure into a LAM code block. Because of this, each code block may contain multiple transactions. It is because of these multiple transactions that the transaction identifier had to be added to the transactional read and write commands. This identifier is just one extra piece of information that needs to be maintained by the run-time system.

An alternative transformation would be to convert the Scheme program into an equivalent continuation passing style (CPS) implementation. With CPS each continuation could be mapped into a code block consisting of exactly one transaction. Under this scheme the transaction identifiers are redundant, since the frame pointer would be sufficient to uniquely identify a transaction.

With CPS, more code blocks would be created, thus a greater potential for parallelism. However, there is a trade off. As the number of code blocks increase, the amount of useful work being done by each code block decreases. Thus the overhead involved with allocating the frame, scheduling, sending messages, and committing becomes harder to amortize. Now that a code block no longer represents a procedure, the environment frame will be located in a separate frame, thus data locality decreases.

By making a code block represent one transaction trade offs have to be made. Simulation is needed to show how expensive these trade offs are.

Garbage collection

There is currently no garbage collection. The mechanisms for providing an efficient distributed garbage collection system for Liquid still need to be worked out.

The garbage collector will need to free up memory which has been allocated from the heap. Besides the heap allocated structures of cons cells and vectors, the garbage collector will need to be able to reclaim frames created by code block invocations. These frames can be collected when there are no more references to their environment entries, or when the frame has been orphaned because of rollbacks.

The orphaned frames can occur when rollback rolls computation back to before the frames allocation, resulting in a new frame being allocated. Computation will proceed using the results from the new invocation of the code block, leaving the previous invocation orphaned.

There are likely to be instances where the orphaned frames still have computation executing relative to them. The garbage collection will have to terminate the computation for these frames by removing any scheduled entries from the LCV and RCV, and then reclaim the frame storage.

Speculative execution

Speculative execution has been shown to be an important method for increasing the amount of available parallelism. However, the problem with speculation is what happens when side-effects occur in speculatively executed code. If the speculatively executed code is determined to be needed, then there is no problem. But, when the code executed is not on the actual computation path, the side-effects need to be undone.

Under Liquid, speculatively executed code containing side-effects does not present a problem. The reason for this is Liquid's transactional model. Only code which is actually a part of the computational path will ever receive the commit. All side-effects by incorrectly executed transactions will not affect the memory system since the commit will never be received.

However, the current LAM does not provide sufficient mechanisms to support speculation. There is no way to specify which speculatively executed code is more promising than another. Nor is there a way to terminate speculatively executed code once it has been determined to be unnecessary.

Multiple commit tokens

At present there is only one commit token which is passed from transaction to transaction. This is to guarantee that all side-effect to shared objects occur as they would in the sequential execution of the program. However, when the compiler can prove that two transaction blocks are data independent, it can allow both of the to commit concurrently.

To allow for simultaneous commits, the compiler needs to be able to send the commit token to more than one successor. And these multiple commits will need to be able to merge back to a single token when the compiler can not prove that concurrent committing is safe.

To send the commit to more than one successor, multiple commit messages would be sent. A thread which synchronizes on the number of outstanding commits would provide a way of merging multiple commits back to a single commit.

Bibliography

André DeHon. Virtual memory and memory viewpoints for distributed systems. Transit Note 82, MIT Artificial Intelligence Laboratory, April 1993.

Thorsten von Eicken, David E. Culler, and Klaus Erik Schauser. TL0 version 2.1: An implementation of threaded abstract machine (draft). Technical report, University of California, Berkeley, August 1991.

Marc Feeley. An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors. Ph.D. thesis, Department of Computer Science, Brandies University, April 1993.

Marc Feeley and James S. Miller. A Parallel virtual machine for Efficient Scheme Compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

Suresh Jagannathan and James Philbin. A customizable substrate for concurrent languages. In ACM SIGPLAN '91 *Conference on Programming Language Design and Implementation*, June 1992.

Suresh Jagannathan and James Philbin. A foundation for an efficient multi-threaded scheme system. In *Proceedings of the 1992 Conference on Lisp and Functional Programming*, June 1992.

Morris J. Katz. Paratran: A transparent transaction based runtime mechanism for parallel execution of scheme. Master's thesis, MIT 1986.

Thomas F. Knight Jr. An Architecture for Mostly Functional Languages, Chapter 19, pages 500-520. MIT Press, 1990.

Monica S. Lam and Robert P. Wilson. Limits of control flow parallelism. In *19th Annual ISCA (ACM)*, pages 46-57, May 1992.

Randy Osborne. Speculative computation in multilisp. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198-208, 1990.

James Philbin. *STING: An Operating System Kernel for Highly Parallel Computing*. Ph.D. thesis, Dept. of Computer Science, Yale University, 1992.

Jonathan Rees and William Clinger, editors. The Revised^4 Report on the Algorithmic Language Scheme. MIT/AI/Memo 848b, November 1991.

Ellen Spertus. Execution of Dataflow Programs on General-Purpose Hardware. Master's thesis, MIT, August 1992.